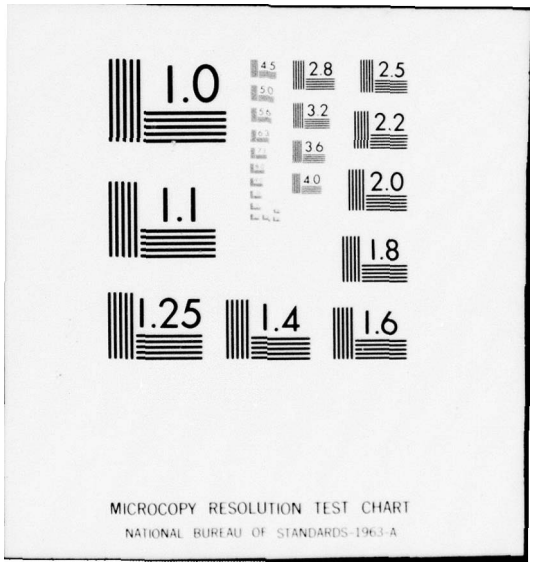
AD-A037 438 WHARTON SCHOOL OF FINANCE AND COMMERCE PHILADELPHIA P--ETC F/G 9/2
ALERTERS ON NETWORK DATABASES.(U)
DEC 76 S F COHEN

UNCLASSIFIED 77-02-07 N00014-75-C-0462 NL

1 OF 1
AD A037438

END

DATE FILMED
4-77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

# ALERTERS ON NETWORK DATABASES

Stanley F. Cohen

77-02-07



A thesis submitted to the Faculty of
The Moore School of Electrical Engineering
in partial fulfillment of the requirements
for the degree of Master of Science in
Engineering (for graduate work in
Computer and Information Sciences)

University of Pennsylvania
Philadelphia, Pennsylvania

December, 1976

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

## REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 77-02-07
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE *(and Subtitle)*: Alerters on Network Databases
5. TYPE OF REPORT & PERIOD COVERED: Final rept.
6. PERFORMING ORG. REPORT NUMBER: 77-02-07

7. AUTHOR(s): Stanley F. Cohen
8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0462

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Decision Sciences Department
University of Pennsylvania/Wharton School
Philadelphia, PA 19104

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: Technical report

11. CONTROLLING OFFICE NAME AND ADDRESS:
Office of Naval Research
Information Systems
Arlington, VA 22217

12. REPORT DATE: 12 /76 (12) Dec 76
13. NUMBER OF PAGES: 59 (12) 58p.

14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*:

15. SECURITY CLASS. *(of this report)*: Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT *(of this Report)*:
Unlimited

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*:

18. SUPPLEMENTARY NOTES:

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*:
Alerters
Network Databases
Event-driven procedures
Demons

408 757

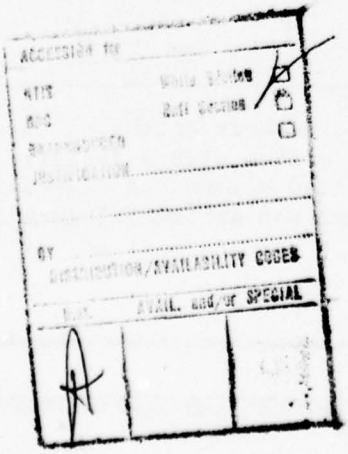
20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*:
This thesis describes a system for alerting on network databases which consists of a simple sharable data management system with a facility for the user to create event-driven procedures called demons. A discussion is included of related work in database systems and artificial intelligence.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Table of Contents





# INTRODUCTION

The thesis falls into two major parts. In the first part I discuss the concept of demons, event-driven procedures which appear to run concurrently with the processes they are monitoring. In the second portion of the thesis I discuss a system for alerting on network databases called SANDL and its precursor, LDEMON.

Some important points that will be examined are the sharability of databases, with accessing and updating functions that are conceived as message handlers, (the requirement of sharability being part of the justification of an alerter system in the first place) as well as the functional framework for the data base accessing and updating functions and their implications, and the modifiability of the schema which gives the user the power to define new record classes and demons at will while the system is running on-line.

Demons should be thought of in the context of whether knowledge is synthesized in terms of goals or in terms of data. At first sight this distinction might seem more appropriate to an artificial intelligence application such as computer vision, but in fact it is a relevant issue for data base technology, where the interest is not in acquiring an understanding of some scene, but in forming an intelligent appraisal of the

current state of affairs represented by the data base. Goal-driven activities in artificial intelligence applications tend to deal with hypothesis formulation, a sort of top-down dividing and conquering of the problem. Most well-structured programs outside the domain of artificial intelligence can be put into this framework. Data or event-driven activities (no distinction between what is and what changes is necessary here) tend to be bottom-up, in the sense that it is the input data which proposes the way in which it should be handled. A good example outside of artificial intelligence is an intepreter, where it is the input that tells the system what to do next. Demons, which monitor their preconditions and take action when these preconditions become true, are one possible implementation of an event-driven system.

Bobrow and Norman (1975) point out that driving an automobile is a paradigm for an event-driven activity. Even though the strategy of a car trip may be planned in advance to have a certain destination and route, the tactics of driving are largely event-driven and concurrent with a host of other activities - looking at the scenery, talking, and so on. Most of the activity of driving follows in response to changes in the data, for which no plans could be made in advance.

How does this distinction relate to databases? Ordinarily we expect a data management system to be preprogrammed to respond to some few types of requests, e.g. what is the name of so-and-so, change the balance of this account in this way, etc. Rarely is a user able to program the system, only to command it. If he should want to observe changes in the data he has no recourse except to make the same query again and again. The object of the research on alerters here reported is to enable users to ask questions like:

"Report the name of any subscriber who changes his address."

or "Give a warning if a ship is too low on fuel to reach a destination."

or "Report if any department has more employees out sick than the average."

or "Take corrective action if any bank balance falls below zero dollars."

These are just a few representative examples of a common management problem. A bank may wish to monitor the profitability and financial health of companies to which it had lent money, or a company may want to keep constant track of size of inventories versus sales.

Alerters provide an answer whenever the manager has the need to dynamically supply the conditions to be monitored.

## Guarded Commands

Dijkstra has presented a new formalism for nondeterministic programs which is particularly well suited to proofs of correctness and termination. He notes that a number of deterministic programs may be mapped into a single nondeterministic program which has the same effect. In order to do this, Dijkstra invents a syntax for the guarded command, which is a list of statements preceded by a Boolean expression. If the Boolean expression is true, the guarded command may be executed, but need not be.

The syntax provides for a repetitive construction and an alternative construction, which are the nondeterministic equivalents of do-loops and if-statements and both of which are guarded command lists, i.e. lists of guarded commands any one of which may be executed if its guard is true. In the alternative construction if any guard is true its guarded command may be executed and at least one of them will be executed. If no guard is true the alternative construction terminates with an error. The repetitive construction on the other hand executes any guarded command whose guard is true until no guard is true, at which point, the repetitive construction terminates successfully. It is this repetitive construction which is important in the

discussion of demons.

Two important ideas can be taken from Dijkstra's article. First, there is the notion that the guarded command, like critical sections in operating systems, can be used to specify that no other activity can interrupt during the execution of the statements in the guarded command. This corresponds to the requirement that demons must be blocked from executing during the middle of an update (unless they are FAIL demons), otherwise the obvious cycling occurs. For example if a demon is monitoring the credits and debits of a balance sheet, there will always be some point during the middle of an update when the credit and debit figures do not correspond, and the demon must be prevented from prematurely taking action.

Secondly, guarded commands provide a formalism for demons that abstract away from specific deterministic implementations. Informally, the set of demons can be taken as guarded commands in the do od repetitive construction which attempts to execute any one the guarded commands until no guard is true.

Demons in Languages for Artificial Intelligence

In PLANNER the basic deductive mechanism is the "theorem", which consists of two parts: a pattern to be matched, and a program. The language supports three types of theorems: "consequent", "antecendent", and "erase". The consequent theorem is closest to usual programming techniques since the pattern to be matched is considered proven if the program of the theorem executes successfully. Essentially, this is little different from the manner in which a program executes successfully only when all its subroutines run to completion. The THGOAL primitive acts somewhat like a CALL statement of FORTRAN or PL/1 with a good deal more sophistication. THGOAL attempts to find its argument (a pattern) in the database, but if the search for an assertion which matches the pattern fails, THGOAL tries to execute consequent theorems which match the pattern until one succeeds and the pattern is "proven".

The antecedent and erase theorems operate in a distinctly different manner. The program for an antecedent theorem is invoked whenever an assertion matching its pattern is asserted. In this case the antecedent theorem is directed by an event, the addition of an assertion, which is the essential characteristic of a demon. Correspondingly, an erase theorem monitors

deletion of assertions which match its pattern and then executes its program. Antecedent and erase theorems are customarily used to add and delete assertions which are implied by the assertion which matches the pattern.

QLISP has a more general method for handling which allows teams of demons to be set up that will be invoked upon any storage or retrieval operation. PLANNER-type antecedent and erase theorems could be built out of such teams of demons.

What the languages for research in artificial intelligence lack in their demon mechanisms are

1. a clear concept of hierarchical record; the basic unit of the database is the assertion, simply a list of assertion type and fields, usually atoms.
2. a high-level concept of demons that observe changes in existing records, not only additions and deletions of assertions from the database.

## Alerter Binding

Alerters, as treated in this thesis, are pairs of condition and program, similar in form to PLANNER theorems or Dijkstra's guarded commands. Morgan (1970) has dealt with the subject of event sequenced programs in some detail, concentrating on the resolution of conflicts caused by attempts to update a variable simultaneously by different programs. The approach taken here differs in that the alerters run conceptually asynchronously and do not require the update to be delayed until the program portion of the alerter has run to completion.

The first task in explaining the alerting mechanism is to sketch the range of possibilities for the object to which the alerter can be bound. I follow the classification of Morgan and Buneman (1975) which divides alerters into several types according to the richness of their properties.

### A. Binding to variables

The most primitive sort of binding which an alerter could have is at the level of variables, that is, simple storage locations. I include this level of alerter binding to emphasize how the more elaborate types of bindings depend on the existence of records, with record classes containing multiple instances of a record type.

B. Simple Alerting

Simple alerting applies to alerters which can be bound to a single record in the database in order to monitor it.

1. monitoring an item, i.e. one or more fields of a record, for example the age of a particular student.

2. monitoring a single record as a whole, for example change to any field of a bank account record.

3. monitoring a field for a record type, for example the age field in any record of the student type.

4. monitoring the addition of a record to a set of records.

5. monitoring the deletion of a record from a set of records.

6. monitoring any field of any record, for example reporting that any personnel record has been modified in any way.

C. Structural Alerting

Alerters can monitor the structure of the database, in other words the relations between records in terms of set ownership. This requires at a minimum that the alerter be bound to an owner record as well as an owned record.

1. Change in set ownership or membership of records. For example a doctor receives a new patient or a company a new supplier.

2. Change in properties of the set as a whole. For example, the number of students in a class exceeds 30.

3. Change in fields of the owned record. This may arise in at least two ways: a doctor may acquire a new patient with measles or a patient already being treated may contract measles.

D. Complex Alerting

Complex alerting covers those types of alerters which require an even more holistic point of view in order to handle the interaction between user and database.

1. Statistical alerters. For example, inform the user if the average bank balance of all accounts changes by more than a given amount. It would be a simple task to plan for the system to keep running counters for average, maximum, minimum, count and other simple statistical quantities, but it is apparent that more complicated calculations will require complete scans of the database.

2. Alerters which monitor transactions over time. For example, inform the user whenever a bank balance drops by more than $10,000 in any 24 hour period. This form of alerting requires that a log be kept of all transactions for the 24 hour period.

3. Pattern recognition. This type of alerter would monitor the creation of a pattern by whatever means that occurs. There would be no distinction between changing of existing records and addition of new ones if they give rise to the desired pattern.

4. Time based alerting. Although this is similar to number 2. the distinction is that the alert need not be signalled immediately, but only within suficient time to be useful. Such alerters could deliver reports on a weekly schedule for example.

5. Monitoring expressions. This form of alerting is an elaboration of the simpler forms of alerting in the direction of providing Boolean combinations of conditions on the alerter. The alerter must therefore be bound to possibly several records at once. For example, the user might want to know whenever a stock had exceeded its previous daily high and sold in greater volume than the previous day.

6. Alerting on the structure of the data base. The issue here is to alert on a chain describable through the schema from possibly multiple owners to a record. It may require bindings to several records at the same time. This is the sort of alerter binding required by requests to monitor say the age of the grandfather of some individual, where the point in the data base may be arbitrarily distant, although reachable.

Binding Time

The issue here is that when the demon is triggered its arguments may no longer be up-to-date if they have been modified since the demon was triggered. If more than one processor (human or computer) has been alerted that, say, a fuel level was too low or some service was required, one of the processors might have taken action before the other, so the second must retest its condition to make sure that the action is still needed. The issue here is to alert on a chain describable through the schema from possibly multiple owners to a record. It may require bindings to several records at the same time. This is the sort of alerter binding required by requests to monitor say the age of the grandfather of some individual, where the point in the data base may be arbitrarily distant, although reachable.

## LDEMON LISP Alerter System
## (and a DAISY interface)

LDEMON is a system consisting of two packages, one for creating and updating a simple data base, and the other for monitoring changes in the data by means of procedures known as demons.

The approach taken by LDEMON has several features of interest to the user. First, the user is able to define new record classes and demons interactively. There is no need to plan demons at the time of creating a new record class, as would be necessary with a compiled system. The only restriction is that a demon should not be created prior to defining all relevant record classes.

A second point is that data bases may be shared among several users, within DAISY, for instance. Any of the users may add new record classes, new demons, new records, or updates to existing records.

Third, even though LDEMON is not at all a production system, since is written in an interpretive language, LISP, and does not handle data bases in auxiliary storage, nevertheless it is an efficient approach to processing data base demons, because the labor involved in monitoring changes is of the order of the number of updates and is independent of the size of the data base.

As a consequence, the approach of LDEMON is particularly useful for large data bases. Finally, the record handling facilities of LDEMON provide a set of primitives written in LISP that can form the basis for building other experimental data base systems, as, for example, David Root has done in writing a relational data base language on top of LDEMON.

## Record Handling Facilities

LDEMON provides means for creating record classes, adding new records, displaying and selecting records, and updating the contents of records.

### Creating a record class

The function used for creating a new class of records is DEF-RECTYPE, with the name of the record for the first argument, followed by all the field (or accessor) names of that record. For example,

```
(DEF-RECTYPE STUDENT SCHOOL AGE)
```

creates a new record class STUDENT with accessors SCHOOL and AGE.

DEF-RECTYPE provides the user with a record class predicate function and one accessor function for each field. In the example here, a new function STUDENT has been created which returns T or NIL depending on whether its single argument is or is not a STUDENT record. In addition, there have been created two accessor functions, SCHOOL, and AGE, which select the SCHOOL and AGE fields of their argument. The record class predicate and the accessor functions all take one argument and may be applied either to a single record or a list of records. In the second case they return a list of results.

Thus, if there is a STUDENT record pointed at by Q, with SCHOOL field PENN and AGE field 20,

```
(STUDENT Q)

returns     T

and

(SCHOOL Q)

returns     PENN
```

Besides creating these auxiliary functions, DEF-RECTYPE defines the record class name as legal for use in adding new records and updating existing ones.

## Adding new records

Two functions are of use in adding new records to an LDEMON "file" (i.e. list of records corresponding to a record class). A "file" is created by adding new records once the record class has been started.

The function GENCONS takes as arguments a record class name followed by values for every field of the record class, in the appropriate order. Accessor or field values may be either alphanumeric (unquoted) or numeric. For example,

(GENCONS STUDENT PENN 21)

creates a new STUDENT record with PENN for the value of SCHOOL, and 21 for the value of AGE.

The effect of GENCONS is to add the new record to the front of the current list of records kept on the RECORD property of the record class name.

Another function, CREATE, allows the user to create one or more identical records . All the arguments used by GENCONS follow an integer for the number of times the same record is to be added to the data base. For example,

(CREATE 1 STUDENT DREXEL 22)

has exactly the same effect as (GENCONS STUDENT DREXEL 22) , while

(CREATE 3 STUDENT TEMPLE 23)

will create three records with TEMPLE for SCHOOL and 23 for AGE.

## Displaying records

The function DISPLAY enables the user to display all records of a given record class or all values of a specified field of a record class. The first argument of

DISPLAY is a record class name and the optional second argument is a accessor or field name. For example, if (GENCONS STUDENT PENN 21) and (GENCONS STUDENT DREXEL 22) have been executed,

(DISPLAY STUDENT)

returns (((STUDENT) DREXEL 22) ((STUDENT) PENN 21))

and (DISPLAY STUDENT AGE)

returns (22 21)

From the description of the record class predicate and accessor functions it should be clear how

(STUDENT (DISPLAY STUDENT))

returns (T T)

and (AGE (DISPLAY STUDENT))

returns (22 21)

since DISPLAY returns the list of records in a record class when applied to a record class name.

## Selecting records

The function RECORD selects a particular record from a record class. The first argument is an integer referring to the reverse order of accession of the record and the second argument is a record class name. For example,

(RECORD 1 STUDENT)

returns ((STUDENT) DREXEL 22)

The user of LDEMON may prefer to keep explicit pointers to records by doing something like

(SETQ FRED (GENCONS STUDENT HAVERFORD 19))

at the time of creating a record.

## Updating records

To change field values, the function SET-VAL is used. The first argument must specify a single record by means of a pointer to the record in an expression like

(EVAL (QUOTE <pointer>))

or a selecting expression which points to a single record. The second argument is a record class name, and the third argument is a field value. For example,

(SET-VAL (RECORD 1 STUDENT) SCHOOL CORNELL)

will change the value of the SCHOOL field to CORNELL.

It should be noted that all records are changed in place, so no new space is acquired by SET-VAL.

# Demons

## Defining new demons

LDEMON adds two new functions (from the user's point of view) and two special keywords to the apparatus already in LISP for defining new functions in order to define demons. This is best illustrated by an example. For instance, assuming that there exists a record class BANK-ACCOUNT, with fields BALANCE and NAME, and the demon is supposed to print the name of any depositor whose bank account balance falls below 200 dollars. The demon to do this can be written as

```
(DEMON OVERDRAWN (X)
 (JCOND ((LESSP (BALANCE X) 200)
           (PRINT (NAME X))
           (PRINC (QUOTE "IS OVERDRAWN"))
))      )
```

As seen from this example, a demon is defined much like a LISP function, with a name, OVERDRAWN, an argument list, (X) , and a function body. The argument list contains only one variable, X , which is bound conceptually to all records of any record class which has both BALANCE and NAME among its fields. It should be emphasized that the binding is not to a variable or an

classes containing multiple instances of a record type.



instance of a record, but to a set of records. In other words, the demon OVERDRAWN should be read as if it constantly monitored all BANK-ACCOUNT records (and the records of any other record class that had BALANCE and NAME fields) and printed the NAME field of any such record whenever its BALANCE field went below 200.

The JCOND construction used here can be thought of as a natural refinement for demons of the LISP COND. Ordinarily, demons are activated only when changes have taken place in the data base. It is both unnecessary and impractical for a demon to perform its action continually as long as its condition is true. Therefore, instead of testing a condition with a COND, LDEMON uses the JCOND to perform the action only when a condition has just become true which was not true before. In the example, this means that the name of a depositor is printed only once whenever his bank account becomes too low. If a COND had been used instead of the JCOND, the name would be printed every time the balance changed as long as it was below 200 dollars.

The demon OVERDRAWN is processed by the DEMON function to form a LISP function called OVERDRAWN, which is then added to the DEMON property of all the field names. The LISP function looks like this:

```
(LAMBDA (X)
 (COND ((AND (LESSP (BALANCE OLD) 200)
             (NOT (LESSP (BALANCE NEW) 200)))
        (PRINT (NAME X))
        (PRINC (QUOTE "IS OVERDRAWN"))
))      )
```

The argument variable X in the first predicate of the JCOND has been replaced by OLD and NEW in the course of transforming the JCOND into an ordinary LISP COND. These two new keywords are introduced to refer to the record before and after it has been updated. OLD and NEW may also be used in the action part of the demon. The old balance could be printed, for instance, by writing

```
(PRINT (BALANCE OLD))
```

In more detail, the DEMON function takes three or more arguments of which the first is the name of the demon, the second is the argument list, which may contain only one variable, and the third and further arguments are any LISP expressions. The third or further arguments must contain an accessor expression which refers to a field of some existing record class, like

```
(NAME X)
```

and the argument of that accessor must be either the variable in the argument list, or OLD, or NEW. OLD or the argument list variable refer to the value of the

field before the update, and NEW refers to the value after the update. For example, if a new record class A has been defined, with fields B and C, the following could be written,

```
(DEMON A1 (X) (PRINT (B X)))
```

which will cause the old value of any B field to be printed whenever it has been updated by a call to SET-VAL.

JCOND

The JCOND (for Just changed COND) is the construction which allows a demon to register a change that has just happened. Its syntax is similar to that of a COND with a single predicate-action pair. However, unlike the COND, the JCOND has only one alternative; if the predicate for this condition is not true, the demon simply does not perform its action.

```
(JCOND (<predicate> <action>))
```

Using the same example as before, and assuming that B is a numeric field, it would be possible to write

```
(DEMON A2 (X) (JCOND ((GREATERP (B X) 7)
                      (PRINT (C NEW)) )))
```

to create a demon which would print the value of the C field (OLD, NEW, or X mean the same here), whenever the

value of the B field has been changed to a value larger than 7 from a value that was not larger. The important point to note here is that a JCOND is converted without change into a COND unless it involves an accessor expression with the demon argument variable as its argument. This is why the predicate is (B X) in the example.

The predicate may be of any complexity, provided it follows these rules, and a sequence of actions may follow the predicate, as in a COND in UCI LISP.

At the present time demons are not explicitly qualified by *record classes; the field names in the* action and condition portions of the demon implicitly restrict the set of record classes. This can cause difficulties if two record classes share a field name, since both record classes will cause a demon to be activated even though only one was intended. However, the user is always able to limit the demon to one record class by using the record class name as a predicate. This was a design decision that could warrant changing for different types of user interaction with the data base.

## The Daisy Interface

LDEMON allows the user to create simplified demons called alerters to observe updates of a data base. The LISP user, however, has available the full power of demons in LISP, and these are essentially anything that can be programmed in LISP, as can be seen from the previous section.

### The ALERT function

The DAISY user needs to know only the ALERT function to specify simple alerters which are triggered by a single condition on some field value in a record. The ALERT function takes as first argument a name for the alerter to be created; as second argument a condition on whether the alerter is to be triggered; and as optional third and further arguments LISP expressions to do whatever actions are required. The condition may be any of the following:

(CHANGE <field name>)
or (<field name> <relation> <field value>)

where <relation> is one of EQUAL, GREATERP, or LESSP. For example,

```
(ALERT SCHOOL-CHANGE (CHANGE SCHOOL)
                     (PRINT AGE))
```

which creates an alerter SCHOOL-CHANGE to print the value of the SCHOOL field in any STUDENT record which has just had its SCHOOL field updated.

If this alerter is compared with the demons of the last section, it will be seen that the demon argument variable, OLD, and NEW are not used. The arguments of the accessors used in the alerter are filled in by ALERT, which creates a call to DEMON as the following example will show.

```
(ALERT HAVER (SCHOOL EQUAL HAVERFORD)
             (PRINT AGE))
```

becomes

```
(DEMON HAVER (X) (JCOND
     ((EQUAL (SCHOOL X) (QUOTE HAVERFORD))
          (DAISYWRITE "ALERT " (QUOTE HAVER))
          (PRINT (SCHOOL NEW))
                                   )))
```

which turns into the function HAVER with definition

```
(LAMBDA (X)
     (COND ((OR (STUDENT X))
                (COND
                  ((AND (EQUAL (SCHOOL NEW)
                               (QUOTE HAVERFORD))
                        (NOT (EQUAL (SCHOOL OLD)
```

```
                                    (QUOTE HAVERFORD))
))
                            (DAISYWRITE "ALERT "
                                    (QUOTE HAVER))
                            (PRINT (SCHOOL NEW))
                        )) )) )
```

The DAISYWRITE expressions which are created by ALERT tell the DAISY user which alerter has been triggered.

## Records and Functions in SANDL

In contrast with the LDEMON system which handles records consisting of data items, the SANDL data management system is based on the semantics behind the DBTG report translated into a more LISP-compatible functional format.

In DBTG there are three basic relations between objects: the one-to-one relation between records and their fields, the one-to-many relations between owner record and the set of records owned, and a many-to-many relation between records which is accomplished by the so-called confluent hierarchy in which a record owned by several records simultaneously in the one-to-may mode is used to embody the many-to-may relation between all the owners.

In the SANDL system the first type of relation is thought of as a function from record to field. The function has the same name as the field. Since the field names are not required to be unique in the schema, the field function is in fact a family of functions from which the appropriate one is chosen according to the record class of the argument.

The second type of relation is treated in two ways: as function from record to owned set with the name of the record field used as the set accessing function. (This corresponds to the downward arrow in the diagram.) It is also treated as a function from an owned record to its owner, in which case the owner record class is the function name. There is a possibility of ambiguity however if a record can be owned be the same record in two different sets. The owner record need not be immediately above the owned record since the system can automatically refer to the schema to find an access chain from owner to set member.

A particularly good point in favor of the functional syntax is how well it matches ordinary English language usage in accessing data. Instead of spelling out an access chain to the desired data object, the user can much more simply use

field ( record (object))

as in NAME (FATHER (X)) which reads as NAME of FATHER of X. While this parallelism between English and functional notation might be painfully obvious, it does pose a question why a more programmatic form of data access by a chain of pointers is not used. Perhaps human cognitive data management is much better suited to dealing with intentional processes rather than extensional ones such as set membership.

The many-to-many relation is a bit trickier to handle within a functional framework. In the doctor-patient relation shown below, where a doctor has many patients and a patient, many doctors all connected through a confluent record CASE, the best that can be done is to return the set of patients for the doctor or the set of doctors for the patient.

In the above discussion, I emphasize that set names are simply fields that access sets. For the most part they are not needed in accessing records except in cases of ambiguity and one-to-many relations.

# Implementation

This portion of the thesis provides the specifications of a system for alerting on network data bases. SANDL is an extension of an earlier system, LDEMON, a data base management system written in LISP with a capability for monitoring changes in the data by means of procedures which appear to be continuously active, called demons.

## Statement of the Problem

The design of SANDL allows demons (also called alerters) to be placed on an updatable network data base management system which has the power of the basic semantic notions detailed in the CODASYL report on DBTG, without any attempt to deal with specifics of implementation. What will be new to LDEMON is the DBTG concept of set ownership and the ability to create demons which monitor changes in the composition of a set.

The general bias throughout this paper has been to avoid as far as possible any need to refer to sets through set names, by an intelligent use of a schema which stores a representation of the hierarchy of record types. Thus, although a set may always be referred to unambiguously through a field of type SET in the owner

record, it is sufficient to supply the owner record and the record type name of the owned set of records, if the relation is unique.

## Data Definition Language

The data definition language (DDL) allows the user to specify what types of records may be placed in the data base, with what fields, and what domain (or type) of field values. In addition, the DDL specifies set ownership and set membership of records. In contrast with the situation in DBTG, the DDL of SANDL is dynamic.

In order to inform the data base management system as well as the demon processor about the current state of the record hierarchy, a special directed graph known as the schema is kept in memory and is updated by the DDL. The schema contains information on all record classes, their field names and range of values, as well as set owner - set member relations. In DBTG, one can naturally construct three distinct sorts of relations between pieces of data; a one-to-one relation between a record and its constituent fields, a one-to-many relation between an owner record and the set of records owned by it, and a many-to-many relation (called a confluent hierarchy) between records which share owned records between them.

The schema is modifiable and means are provided for displaying it to the user. Saying that the schema is modifiable means that new record types can be added to the data base while the system is running and additional fields can be declared for already existing record types. Changes in the schema will of course be constrained by the necessity of not destroying entire sets of records by destroyin g their record type.

Creating new record types

In LDEMON a new record type is created by a call to the function DEF-RECTYPE of the form

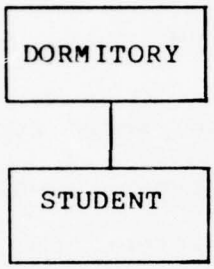
(DEF-RECTYPE STUDENT SCHOOL AGE)

in order to define a new record type STUDENT with fields SCHOOL and AGE. SANDL replaces this form of the function call by adding arguments for the owner record types and the type of each field.
For example,

(DEF-RECTYPE STUDENT (DORMITORY) SCHOOL CHARACTER
AGE NUMBER )

where DORMITORY is the record type of the only owner record, SCHOOL is declared of type CHARACTER, and AGE of type NUMBER.

DEF-RECTYPE adds the STUDENT record type to the data base schema and places this record in the relationship of set member to the owner record class DORMITORY. In DBTG terms,




Furthermore, selector functions are created corresponding to each of the fields SCHOOL and AGE. Unlike in LDEMON, the function STUDENT, also created by DEF-RECTYPE, is not a record type predicate but a selector function that yields the set of students when applied to a DORMITORY record.

The syntax of the DEF-RECTYPE function call is

(DEF-RECTYPE <record type>
<list of owner record types>
<field and type>* )

and <field and type> is

<field> <type>

where <type> is chosen from the list

NUMBER

CHARACTER (with no length qualification)

SET

which may be abbreviated to NUM, CHAR, and SET.

In the example below, the record type DORMITORY is defined with some field of type SET which is to own the STUDENT records. This is an intentional departure from DBTG and combines the relationship of data-items to a record with that of sets owned by the record. If DORMITORY is considered to be a top-level record type in the schema, i.e. a member of no set owned by *any other* record, its owner is simply the null set of record types, NIL. In this case the record type DORMITORY is defined by

```
(DEF-RECTYPE DORMITORY NIL RESIDENTS SET
                           NAME CHARACTER)
```

## Data Management Language

The data management language (DML) allows the creation, deletion, and modification of records contained in sets owned by other records. In addition, functions are to be provided that select subsets of records that meet specified conditions.

### Adding new records

To add a new record to the data base requires that the record type and the owner record be supplied, together with the appropriate field values for each field defined for the record type. In other words, the arguments of the function to do this, INSERT, are the same as those for GENCONS in LDEMON, except for the addition of the owner record as an argument. For example, if DOC1 is a particular DOCTOR record, where the DOCTOR record is defined as owning a set of records, and PATIENT has been defined by

```
(DEF-RECTYPE PATIENT (DOCTOR) AGE NUMBER
                              SEX CHARACTER
                              NAME CHARACTER )
```

then

```
(INSERT PATIENT DOC1 20 MALE JONES  )
```

will insert a PATIENT record into the set of patients

owned by the DOC1 record. DOC1 serves as a unique identifier of some particular DOCTOR record. In this implementation it is a pointer to the specified record, but it can be thought of as an identifying key. The value of the INSERT function is the newly created record. If the record to be inserted belongs to a top-level record type then the function needs no owner record argument.

The syntax of INSERT is

(INSERT <record type> <field value>* )

or

(INSERT <record type> <owner record>
<field value>* )

where there must be as many field values as there are fields in the record type and they must agree in type and order with their definitions at the time of defining the record type.

Deleting records

A record can be deleted from the data base by the function DELETE with value NIL. When a record is deleted it is removed from all sets to which it belongs. The syntax of the function call is simply

(DELETE <record>)

Updating records

Values of NUMBER and CHARACTER fields may be changed by the function SET-VAL, with the syntax

(SET-VAL <record> <field name> <field value>)

where <record> is any record in the database with the data item <field name>. No owner record is needed. A further function, SET-VALUES, changes all the fields of a record, leaving those with <field value> equal to '*' unchanged.

(SET-VALUES <record > <field value>* )

SET-VALUES must have as many field values as the number of fields in the record type.
SET-VALUES is provided as a convenience for changing several fields of a record at the same time.

Moving records between sets

In order to move a single record from one set to another, as for example transferring a case from one doctor to another, the function TRANSFER is used, with the syntax

(TRANSFER <record> <from owner record>
<to owner record> )

Selecting records from a set

Three functions to select a subset of records or apply a function to all elements of a set are SELECT, SELECT-ONE, and MAPSET. Their syntax is

(SELECT <predicate> <set>)
(SELECT-ONE <predicate> <set>)
(MAPSET <function> <set>)

where <set> is a set of records, <predicate> is a predicate of one argument that evaluates to T or NIL when applied to a single record, and <function> is a function or lambda expression of one argument. For example, if SOMESET is a set of records with a field NAME of type CHARACTER,

(SELECT (QUOTE (LAMBDA (X) (EQ (NAME X)
(QUOTE SMITH))))
SOMESET )

will return as its value that subset of SOMESET whose NAMEs are SMITH.

SELECT-ONE returns that unique record which satisfies the predicate; if more than one exists, the value returned is NIL. The main use for SELECT-ONE is to retrieve items from record types with unique keys.

The function MAPSET applies the function to every record in the set and returns NIL. It could be used for its effect, changing the value of some field for every record in a set, for example.

By applying these functions to sets, explicit loops become unnecessary for traversing a data base. This should be compared with the non-procedural approach taken with the HI-IQ retrieval language in Rob Gerritsen's dissertation, which creates an access path for finding the requested data. A simple example using the PATIENT record type defined above is to find the ages of all male patients of a doctor whose record is DOC1. All that is needed is

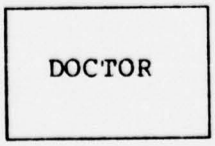
```
(AGE (SELECT (QUOTE (LAMBDA (X)
                     (EQ (SEX X)
                     (QUOTE MALE))))
             (PATIENT DOC1)))
```

## Demons

The ideas presented here go beyond those of LDEMON in two principal ways. Demons are used to monitor records within sets owned by other records, and the scheduling of the processing of demons is more elaborate.

In LDEMON, demons are used to take specific action whenever any member of a record type has its fields modified. SANDL adds the ability of demons to watch for changes in the set of records owned by a given record type. In other words, demons in LDEMON are similar to the IFMODIFIED demons of SANDL, while IFADDED and IFREMOVED demons are completely new.

In LDEMON a demon has a single argument variable which is bound to whatever record triggers the demon. In general, demons monitoring sets bind at least two arguments, one for the record in the set, the second for the record owning the set. The one exception is that top-level records are not owned by any record, so the DBTG schema




corresponds to a demon argument list

(DOCTOR X)

The decision was made in LDEMON to execute the action of a demon before the update which triggered the demon. For demons used as alerters (i.e. without any effect on the data base) this seems to have been a bad choice, since the update itself is delayed until all relevant demons have been checked and those triggered have been executed. SANDL therefore distinguishes two types of demons: what will be called fail demons, and non-fail demons. A fail demon will be activated immediately upon being triggered, while a non-fail demon waits until after a change to the data base has taken place. The fail demons can be used to send messages that, say, an illegal update has been attempted, or to enforce data integrity, for example by preventing any field from changing to a value of the wrong type.

Demon syntax

The syntax for creating a demon is

(<demon type> <demon name> <argument list> <body>)

or

(<demon type> <demon name> FAIL <argument list> <body>)

where <demon type> is IFADDED, IFREMOVED, or IFMODIFIED, and <body> is any sequence of LISP expressions.

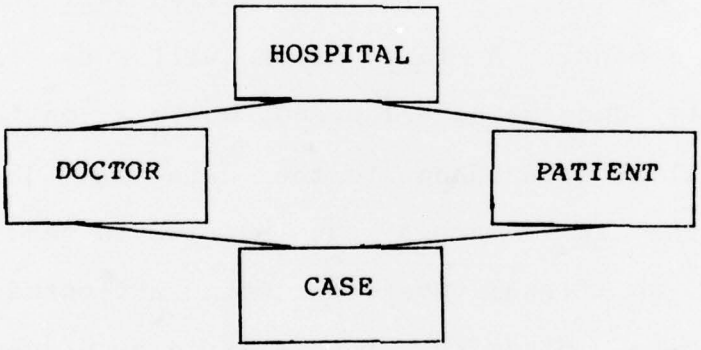
The <argument list> is either

(<record class> <variable name> )

or

(<record class> <variable name> <argument list>* )

To illustrate the use of argument lists compare the DBTG structure




with the argument list

(CASE U (DOCTOR V (HOSPITAL W))
(PATIENT X (HOSPITAL Y)))

Note the two separate bindings of HOSPITAL records to the variables W and Y. The reason for this is that there is no way to exclude the possibility of two different HOSPITAL records leading through separate ownership chains to a particular CASE record. If it is essential that records X and Y be the same, this must be tested in the body of the demon.

The lowest record class in the schema is placed first in the argument list, so the schema is to be read upwards in writing the argument list.

JCOND

The JCOND construction for changes which have just occurred is carried over from LDEMON in the IFMODIFIED demon. Only the COND (the ordinary LISP "IF-THEN-ELSE" construction) is meaningful for IFADDED and IFREMOVED demons, since in these cases there is no distinction to be made between old and new versions of the record.

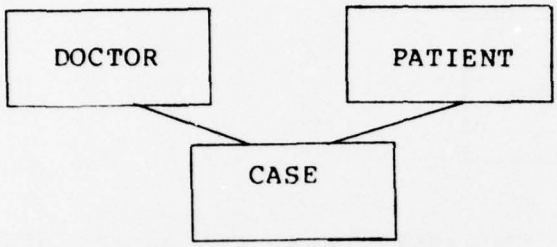
For the IFMODIFIED demon the convention for distinguishing old versus new versions of a record is to prefix the variable with '-' for old and '+' for new. In this way

```
(AGE -X)
```

would stand for the value of the AGE field of the record X before an update. If there is no ambiguity the '+' and '-' prefixes are not needed and the JCOND always uses the unadorned variable in its test.

Some examples of demons

If the DBTG schema is




CASE has a field DIAGNOSIS, and DOCTOR has a field NAME. An expression to print the name of any doctor who has a patient who contracts measles is:

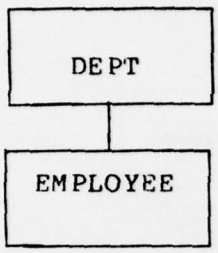
```
(IFMODIFIED M1 (CASE X (DOCTOR Y) )
        (JCOND ((EQ (DIAGNOSIS X) (QUOTE MEASLES))
                (PRINT (NAME Y)))  ))
```

However, to print the name of any doctor who gets a patient who has measles is:

```
(IFADDED A1 (CASE X (DOCTOR Y))
        (COND ((EQ (DIAGNOSIS X) (QUOTE MEASLES))
                (PRINT (NAME Y))) ))
```

A more complicated example is to report the name of any department if the proportion of female staff falls below 25%.

In this case the schema is




DEPT has fields NUM-STAFF and PROPORTION-FEMALE, and EMPLOYEE has a field SEX. Then the problem is solved by means of three demons.

First, the table below gives the changes to PROPORTION-FEMALE (represented by P) and NUM-STAFF (represented by N).

| | Employee | |
|---|---|---|
| | Female | Male |
| Add | P := ((P*N)+1)/(N+1) | P := (P*N)/(N+1) |
| | N := N+1 | N := N+1 |
| Remove | P := ((P*N)-1)/(N-1) | P := (P*N)/(N-1) |
| | N := N-1 | N := N-1 |

```
(IFADDED A2 (EMPLOYEE X (DEPT Y))
   (COND ((EQ (SEX X) (QUOTE FEMALE))
          (SET-VAL Y PROPORTION-FEMALE
             (DIVIDE (ADD1  (TIMES
                               (PROPORTION-FEMALE Y)
                               (NUM-STAFF Y)))
```

```
                    (ADD1  (NUM-STAFF Y)))))
          (T (SET-VAL Y PROPORTION-FEMALE
               (DIVIDE  (TIMES
                          (PROPORTION-FEMALE Y)
                          (NUM-STAFF Y)))
                        (ADD1  (NUM-STAFF Y)))))
    (SET-VAL Y NUM-STAFF (ADD1 (NUMSTAFF Y))))

(IFREMOVED R2 (EMPLOYEE X (DEPT Y))
    (COND ((EQ (SEX X) (QUOTE FEMALE))
           (SET-VAL Y PROPORTION-FEMALE
              (DIVIDE (SUB1  (TIMES
                          (PROPORTION-FEMAI  Y)
                          (NUM-STAFF Y)))
                        (SUB1  (NUM-STAFF Y)))))
          (T (SET-VAL Y PROPORTION-FEMALE
               (DIVIDE  (TIMES
                          (PROPORTION-FEMALE Y)
                          (NUM-STAFF Y)))
                        (SUB1  (NUM-STAFF Y)))))
    (SET-VAL Y NUM-STAFF (SUB1 (NUMSTAFF Y))))

(IFMODIFIED M2 (DEPT W (UNIVERSAL X))
      (JCOND ((LESSP (PROPORTION-FEMALE W) .25)
              (PRINT (NAME W))  ))
```

This example shows how demons can be used to monitor average, maximum, minimum, and count of a set. IFADDED and IFREMOVED demons must update fields in the owner record and an IFMODIFIED demon watches for changes in these fields.

## REFERENCES

1. Becker J.D. "Reflections on the Formal Description of Behavior." in Representation and Understanding: Studies in Cognitive Science, ed. Bobrow, D.G. and Collins, A., Academic Press, N.Y. 1975.

2. Bobrow, D.G. and Brown, J.S. "Systematic Understanding: Synthesis, Analysis, and Contingent Knowledge in Specialized Understanding Systems." in Representation and Understanding: Studies in Cognitive Science, ed. Bobrow, D.G. and Collins, A., Academic Press, N.Y. 1975.

3. Bobrow, D.G. and Norman D. A. "Some Principles of Memory Schemata." in Representation and Understanding: Studies in Cognitive Science, ed. Bobrow, D.G. and Collins, A., Academic Press, N.Y. 1975.

4. Bobrow, D.G. and Raphael B. "New Programming Languages for Artificial Intelligence Research." Computing Surveys 6, 3 (September 1974).

5. CODASYL, CODASYL Data Base Task Group, April 1971 Report.

6. Cohen, S.F. "SANDL: A System for Alerting on

Network Databases in LISP." Working Paper 76-05-07, Dept. of Decision Sciences, The Wharton School, University of Pennsylvania, May 1976.

7. Date, C.J. An Introduction to Database Systems, Addison Wesley, Reading, Mass. 1975.

8. Dijkstra, E.W. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs." Comm. ACM 18, 8 (August 1975), 453-457.

9. Gerritsen, Rob "Understanding Data Structures," PhD Thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1975.

10. Morgan, H.L. "Event sequenced programming." Technical Report No. 119. Dept. of Operations Research. Cornell University, Ithaca, N.Y., September 1970.

11. Morgan, H.L. "An Interrupt Based Organization for Management Information Systems." Comm. ACM 13, 12 (December 1970).

12. Morgan, H.L. and Buneman, O.P.B. "Alerting in Database Systems: Concepts and Techniques." Working Paper 75-12-02, Dept. of Decision Sciences, The Wharton

School, University of Pennsylvania, June 1976.

13. Sussman G. and McDermott D. "Why Conniving is better than Planning." MIT Artificial memo 255A (April 1972).

14. Sussman G.J. and Winograd T. "Micro-Planner Reference Manual," Project MAC Report, Massachusetts Institute of Technology, Cambridge, Mass., 1972.

DISTRIBUTION LIST

Department of the Navy - Office of Naval Research

Data Base Management Systems Project

Defense Documentation Center (12)
Cameron Station
Alexandria, VA 22314

Office of Naval Research (6)
Arlington, VA 22217

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, Illinois 60605

New York Area Office
715 Broadway - 5th Floor
New York, NY 10003

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, DC 20380

Office of Naval Research
Code 458
Arlington, VA 22217

Mr. E. H. Gleissner
Naval Ship Research and
Development Center
Computation & Mathematics Dept.
Bethesda, MD 20084

Mr. Kim B. Thompson
Technical Director
Information Systems Division
(OP-911G)
Office of Chief of Naval Operations
Washington, DC 20350

Professor Omar Wing
Columbia University
Dept of Electrical Engineering
and Computer Science
New York, NY 10027

Office of Naval Research (2)
Information Systems Program
Code 437
Arlington, VA 22217

Office of Naval Research
Code 1021P Branch Office, Boston
495 Summer Street
Boston, MA 02210

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106

Naval Research Laboratory (6)
Technical Information Division
Code 2627
Washington, DC 20375

Office of Naval Research
Code 455
Arlington, VA 22217

Naval Electronics Laboratory Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152

Captain Grace M. Hopper
NAICOM/MIS Planning Branch
(OP-916D)
Office of Chief of Naval Operations
Washington, DC 20350

Bureau of Library and
Information Science Research
Rutgers - The State University
189 College Avenue
New Brunswick, NJ 08903
Attn: Dr. Henry Voos